Laws of Object-Orientation with Reference Semantics

Leila Silva[1], Augusto Sampaio[2], and Zhiming Liu[3]

[1] Departamento de Ciência da Computação - Universidade Federal de Sergipe
CEP 49000-100 São Cristóvão - SE - Brazil
leila@ufs.br
[2] Centro de Informática - Universidade Federal de Pernambuco
CEP 50740-540 Cidade Universitária - Recife - PE - Brazil
acas@cin.ufpe.br
[3] International Institute for Software Technology
P.O.Box 3058, Macao SAR - China
Z.Liu@iist.unu.edu

Abstract. Algebraic laws have been proposed to support program transformation in several paradigms. In general, and for object-orientation in particular, these laws tend to ignore possible aliasing resulting from reference semantics. This paper proposes a set of algebraic laws for object-oriented languages in the context of a reference semantics. Soundness of the laws is addressed, and a case study is also developed to show the application of the proposed laws for code refactoring.

1 Introduction

Object-oriented programming is largely used for potentially increasing the productivity of software development, and there is a great interest on formalisms that allow reasoning about object-oriented programs.

Several approaches to formalise the semantics of object-oriented languages have been proposed. For example, Abadi and Leino [1] introduce a logic for reasoning about object-oriented programming and prove its soundness. Müller and Poetzsch-Heffter [2] introduce some general techniques for object-oriented program verification. Cavalcanti and Naumann [3] propose an object-oriented language, ROOL, similar to Java, whose semantics is based on weakest preconditions and adopts the copy mechanism. He et al [4] propose rCOS, a reference semantics for an object-oriented language, based on the Unifying Theories of Programming (UTP) [5]. The logic of rCOS is a conservative extension of standard predicate logic [5].

Many paradigms have benefited from algebraic programming laws, among them imperative programming [6], functional programming [7], logic programming [8] and concurrency [9]. These laws state properties that relate programming constructs. Nevertheless, programming laws for the object-oriented paradigm are not yet well established. Laws for small grain object-oriented constructs have been considered in, for example, [10]. The work of Borba et al [11] is a significant contribution in this direction. They propose laws of commands and of classes for

ROOL, therefore restricted to copy semantics. Cornélio [12] proves all the laws proposed in [11] and shows how these laws can be used to formalize some of the refactorings informally proposed by Fowler [13]. He et al [4] also propose a set of laws for an object-oriented language, in the context of rCOS. These laws are very similar to the ones presented in [11] but the proofs in the context of a reference semantics are not presented. Furthermore, laws valid exclusively in the context of reference semantics do not seem to have been investigated.

In this paper we present laws for object-oriented languages considering a reference semantics. We consider laws that are valid both for copy and reference semantics, revise some laws that are valid for copy but not for reference semantics, and propose new laws that hold only for reference semantics. As far as we are aware, this is an entirely new contribution. In particular, we use the rCOS semantics to prove soundness of each proposed law, as illustrated in this paper and more detailed in [14]. We also develop a simple case study to show the applicability of the proposed laws for improving code structure.

This paper is organized as follows. Section 2 summarises the rCOS semantics. Section 3 briefly discusses copy versus reference semantics and introduces some laws valid exclusively in the context of reference semantics. Applications of the proposed laws related to code refactoring are the subject of Sect. 4. Section 5 presents some final considerations and directions for future work. Finally, in Appendix A we illustrate the style for proving the proposed laws.

2 The rCOS Approach

To prove soundness of the laws introduced in this paper, we need to use a semantics for object-oriented languages that considers references. We adopt rCOS [4] and this section summarizes some relevant aspects of this formalism.

2.1 Syntax of rCOS

In rCOS an object system (or program) S is of the form `Cdecls` • `Main`, where `Cdecls` consists of a class declaration section and `Main` a main method. The main method is a pair `(externalvar, c)`, where `externalvar` is a finite set of global variables and `c` is a command (the program control flow). The declaration section is a finite set of class declarations $\texttt{cdecl}_1\texttt{;cdecl}_2\texttt{; ... ;cdecl}_k$, where each class declaration $\texttt{cdecl}_i$ has the form given in what follows.

```
[private] class M [extends N] {
          [private   T11 a11 = d11, ..., T1r a1r = d1r;]
          [protected T21 a21 = d21, ..., T2s a2s = d2s;]
          [public    T31 a31 = d31, ..., T3t a3t = d3t;]
          [method m1(V11 x1; V12 y1; V13 z1){c1};...
                  mℓ(Vℓ1 xℓ; Vℓ2 yℓ; Vℓ3 zℓ){cℓ};]}
```

The square brackets mean that the enclosed term is optional. A class can be declared as `private` or `public`, but only public classes or a primitive type can

be used as global variable types in `Main`. Attributes can be `private`, `protected` or `public` and a method declaration declares its value parameters (V_{i1} x_i), result parameters (V_{i2} y_i), value-result parameters (V_{i3} z_i) and body (c_i). All methods are public. rCOS supports the following commands:

$$c ::= SKIP \mid CHAOS \mid var\ T\ x\ [=e] \mid end\ x \mid c_1;c_2 \mid c_1 \lhd b \rhd c_2 \mid c_1 \sqcap c_2 \mid b * c \mid le.m(e_1;e_2;e_3) \mid le := e \mid C.new(le)$$

The command `SKIP` does nothing and terminates successfully, whereas `CHAOS` is the most nondeterministic command. Variables can be declared with `var` and undeclared with `end`. Variables may include an optional initialization. Sequential composition (;), conditional choice ($\lhd \rhd$), non determinism ($\sqcap$) and iteration (*) are provided. The arguments of a method call `le.m` obey value, result and value-result parameter passing mechanisms, respectively. The term `le` stands for an expression that can appear on the left-hand side of an assignment. Such expressions are called *left-expressions* and obey the form `le ::= x | self | le.a`, where `x` is a variable and `a` an attribute of `le`. An expression `e` is of the form `e ::= x | a | null | self | e.a | (C)e | f(e)`, where `null` is a null reference, `self` is used to denote the active object, `e.a` is the attribute `a` of `e`, `(C)e` is type casting and `f` is a built-in operation for a primitive type. The term e_1 is a list of expressions, whereas e_2 and e_3 are lists of left-expressions in the form `x` or `le.a`.

2.2 Semantics of rCOS

In rCOS, the semantics of a program is the combination of its static and dynamic semantics. The static semantics comprises a class declaration and a declaration section and is captured by the *structural variables*, denoted Ω, which do not change during the execution of the program. These variables record the set of names, denoted as *cname*, of the private and public classes, the set of private, protected and public attributes of a given class, the direct superclass of a class and a mapping $op(C)$ that associates with each class C the list of methods of C.

The dynamic semantics addresses commands and the entire program. It uses dynamic variables, which comprise program variables, a variable *visibleattr* (that records the visible attributes) and a variable π that represents the system configuration.

Program variables include global variables, declared in the main method, and local variables (denoted *localvar*), declared by local variable declaration commands. Local variables include *self* (the current active object) and parameters of methods. The union of global and local variables is denoted by *Var*.

As method calls may be nested, *self* and parameters may be declared a number of times with possible different types before they are undeclared. Thus, a local variable x is represented by a sequence of declared types x : *TypeSeq*, *TypeSeq* ::= $\langle T_1, T_2, \ldots, T_n \rangle$. The declared type of x is retrieved by the function dtype(x). The term $\overline{x}$ is used to denote the value of a local variable x, which is in fact a sequence of values, the head representing the current value of x, denoted by value(x).

A value is either a member of a primitive type or is an object identity in *REF* with its dynamic typing information. *REF* is a set of object identities (or

references) and includes *null*. For a value $v = \langle r, C\rangle$, ref(v) denotes the reference r of v and type(v) the current type C of v.

An object o is either the special object *null* or a structure $\langle r, C, \sigma\rangle$, where r, C and σ are, respectively, the reference, the current type and the state of o, the latter denoted state(o), mapping each attribute a visible in C into a value of the declared type of a.

The visibility of attributes is captured by a dynamic variable *visibleattr* that records the set of attributes visible to the command under execution. The value of *visibleattr* defines the current execution environment.

The value of the system configuration π is the set of objects created so far. When a new object is created or the value of an attribute of an existing object is modified, the system configuration changes. We use $\pi(C)$ to denote the set of existing objects of class C.

In the UTP a program is represented by a design $D = (\alpha, P)$ where α denotes the set of variables (the alphabet) of the program and P is a predicate of the form

$$p(x) \vdash R(x, x') \stackrel{def}{=} (ok \wedge p(x)) \Rightarrow (ok' \wedge R(x, x'))$$

where $x, x' \subseteq \alpha$ stand for the initial and final values of a variable; p is the precondition, defining the initial states; R is the postcondition, relating the initial and final states; ok, ok' describe initiation and termination of the program and do not appear in expressions or assignments of the program texts. Designs can be framed and the domain of designs is closed under sequential composition, choice and iteration. Refinement relation between design is defined as logic implication.

As rCOS is based on the UTP, the semantics of a command is defined using the dynamic variables. Each command c has its well-defined conditions $\mathcal{D}(c)$, as part of the precondition of the semantics of c; the semantics of c, denoted $[\![c]\!]$, has the form of $\mathcal{D}(c) \Rightarrow (p \vdash R)$. In what follows we give the semantics of local declaration and undeclaration, assignments and method call. The semantics of other constructors can be found in [4]. In the following expressions the symbol $\preceq$ stands for a subclass relation and $\{v\}$ stands for a framed element v, whose value may be altered by the execution of the command currently defined.

Local variables are represented as sequences of declared types and values. When a variable is declared the corresponding sequences are modified to add new elements in their heads. The command *var* $T\ x = e$ declares a variable and initializes it. The command *end* x terminates the current scope of variable x. Thus, the heads of the sequences of x are removed. The semantics of both commands are given in what follows.

$$\begin{aligned}
\mathcal{D}(var\ T\ x = e) &\stackrel{def}{=} (x \in localvar) \wedge \mathcal{D}(e) \wedge \\
&\quad (\text{type}(e) \in cname \Rightarrow \text{type}(e) \preceq T) \\
[\![var\ T\ x = e]\!] &\stackrel{def}{=} \{x\} : \mathcal{D}(var\ T\ x = e) \vdash (\overline{x}' = \langle value(e)\rangle.\overline{x}) \wedge \\
&\quad TypeSeq'(x) = \langle T\rangle.TypeSeq(x).
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}(end\ x) &\stackrel{def}{=} (x \in localvar) \\
[\![end\ x]\!] &\stackrel{def}{=} \{x\} : \mathcal{D}(end\ x) \vdash (\overline{x}' = \text{tail}(\overline{x})) \wedge \\
&\quad TypeSeq'(x) = \text{tail}(TypeSeq(x)).
\end{aligned}$$

An assignment has two forms, depending on whether le is a variable x or has the form $le.a$. In the first case, the current value of the variable is changed. In the second case, the value of an attribute a of object le is modified.

$$\begin{array}{lll}
\mathcal{D}(le := e) & \stackrel{def}{=} & \mathcal{D}(le) \wedge \mathcal{D}(e) \wedge (\mathrm{type}(e) \in \mathit{cname} \Rightarrow \mathrm{type}(e) \preceq \mathrm{dtype}(le)) \\
[\![x := e]\!] & \stackrel{def}{=} & \{x\} : \mathcal{D}(x := e) \vdash (\overline{x}' = \langle \mathrm{value}(e) \rangle . \mathrm{tail}(\overline{x})) \\
[\![le.a := e]\!] & \stackrel{def}{=} & \{\pi(\mathrm{dtype}(le))\} : \mathcal{D}(le.a := e) \vdash \\
 & & (\pi(\mathrm{dtype}(le))' = \pi(\mathrm{dtype}(le)) \uplus \\
 & & \{o \oplus \{a \mapsto \mathrm{value}(e) | o \in \pi \wedge \mathrm{ref}(o) = \mathrm{ref}(le)\}\}).
\end{array}$$

The overriding ($\oplus$) of an object $o = \langle r, M, \sigma \rangle$ to record the new value d of attribute a is defined as $o \oplus \{a \mapsto d\} = \langle r, M, \sigma \oplus \{a \mapsto d\} \rangle$. For a set $S \subseteq O$ of objects, $S \uplus \{\langle r, m, \sigma \rangle\} \stackrel{def}{=} (S \backslash \{o | \mathrm{ref}(o) = r\}) \cup \{\langle r, M, \sigma \rangle\}$.

Notice that this definition requires dynamic type matching and the type of the object needs to be consistent with the declared type. When le is x, the new value of x is now the head of the sequence of values, replacing the old value of x in the current scope. In the other case, the attached object is found in the system configuration π and its state is modified accordingly.

For a method signature $m(V_1\ x;\ V_2\ y;\ V_3\ z)$ let ve, res and vre be lists of expressions representing the arguments passed by value, result and value-result mechanisms. The semantics of $le.m(ve;\ re;\ vre)$ is as follows.

$$\begin{array}{lll}
\mathcal{D}(le.m(ve;re;vre)) & \stackrel{def}{=} & \mathcal{D}(le) \wedge \mathrm{type}(le) \in \mathit{cname} \wedge (le \neq \mathit{null}) \wedge \\
 & & \exists N \in \mathit{cname}.(\mathrm{type}(le) \preceq N) \wedge \\
 & & \exists (m \mapsto (V_1\ x;\ V_2\ y;\ V_3\ z,\ c_1) \in \mathrm{op}(N)) \\
[\![le.m(ve;re;vre)]\!] & \stackrel{def}{=} & (\mathcal{D}(le.m(ve;re;vre)) \Rightarrow \exists C \in \mathit{cname}.(\mathrm{type}(le) = C) \wedge (\\
 & & [\![\mathit{var}\ V_1\ x = ve;\ V_2\ y;\ V_3\ z = vre]\!]; \\
 & & [\![\mathit{var}\ C\ \mathit{self} = le]\!]; [\![\mathrm{Execute}(C.m)]\!]; [\![re,\ vre := y,\ z]\!]; \\
 & & [\![\mathit{end}\ \mathit{self},\ x,\ y,\ z]\!]).
\end{array}$$

If method m is not declared in C but in a superclass N of C then

$$[\![\mathrm{Execute}(C.m)]\!] \stackrel{def}{=} [\![\mathrm{Execute}(M.m)]\!],$$

where M is the immediate superclass of C and $M \preceq N$. Given a class A and a method m the semantics of Execute($A.m$) comprises setting the execution environment, then executing the body of the method and finally resetting the environment. Thus, $[\![\mathrm{Execute}(A.m)]\!] \stackrel{def}{=} \mathrm{Set}(A); \mathrm{body}(A.m); \mathrm{Reset}$, where Set($A$) sets the variable *visibleattr* with the set of attributes visible in class A, that is, attributes of A, public and protected attributes inherited from superclasses of A, and public attributes of public classes. Reset sets the variable *visibleattr* to the set of attributes that are accessible by the main program, that is, the public attributes of public classes. The term body(A, m) is the body c of method m. The prefix *self* is added to each attribute and method in c, to guarantee that the environment is set

correctly when a nested method call that may change the execution environment is completed.

Based on the semantics of a class declaration, $[\![Cdecls]\!]$, and on the semantics of commands, $[\![c]\!]$, the semantics of a program is defined as

$$[\![Cdecls \bullet Main]\!] \stackrel{def}{=} \exists \Omega, \Omega', internalvar, internalvar' \bullet ([\![Cdecls]\!]; Init; [\![Main]\!]),$$

$$Init \stackrel{def}{=} \mathcal{D}(Cdecls) \vdash visibleatr' = \emptyset \wedge \pi' = \emptyset \quad \wedge \bigwedge_{x \in Var} (\overline{x}' = \langle\rangle \wedge Typeseq = \langle\rangle),$$

$$[\![Main]\!] \stackrel{def}{=} \mathcal{D}(Main) \Rightarrow [\![c]\!],$$

where *internalvar* is the set of variables in *localvar*, π and *visibleattr*, *Init* corresponds to the initialization of variables, and $Main$ is the main program.

3 Laws

The main difference between copy and reference semantics is the possibility of aliasing in the latter. Aliasing can occur through assignments and parameter passing. Aliasing can cause a range of difficult problems within object-oriented programs, because a referring object can change the state of the aliased object, implicitly affecting all the other referring objects [15].

A comprehensive set of laws of an object-oriented language based on copy semantics has been proposed in [11]. This work distinguishes command laws and laws for classes. As aliasing is related to assignment and parameter passing, the laws for classes proposed in [11] are also valid for reference semantics. These laws deal, among other constructs, with changing visibility of attributes (from private to protected and from protected to public), introducing new class declarations, removing unused classes, introducing fresh attributes, removing unused attributes, introducing new methods, removing redundant or unused methods, introducing inheritance, and moving attributes and methods to a superclass or a subclass.

Nevertheless, some command laws are not valid in the context of reference semantics, and here we propose adaptations of such laws for this new context (Sect. 3.1). Moreover, we propose new laws valid only in the context of a reference semantics (Sect. 3.2). We briefly address soundness by presenting the proof of one of these laws in Appendix A, using the rCOS semantics, given in Sect. 2.

An important point to notice is that we consider reference semantics assuming that expressions are side-effect free. Furthermore, we follow the design of Java that does not allow pointer manipulation, and attaches a copy semantics to assignments involving variables of primitive types.

Before presenting new laws, we reproduce here laws 1 (var elim), 2 (var final value) and 3 (;-◁▷ distrib), valid for both copy and reference semantics, as they are used in the development of the case study of Sect. 4.

We say that a variable in a command c_1 is *free*, denoted free(c_1), if it is not declared in the local block under consideration, and *bound* otherwise. Similarly, we use free(e) for the free variables of an expression e. If a declared variable is never used, its declaration has no effect.

Law 1 (var elim) `(var T x; c; end x) = c, if x` $\notin$ `free(c).`

There is no point in assigning to a variable at the very end of its scope. Moreover, evaluation of a condition is not affected by what happens afterwards, and therefore sequential composition distributes leftwards through a conditional. These facts are captured by the next two laws, respectively.

Law 2 (var final value) `(var T x; c; x := e; end x) = (var T x; c; end x)`

Law 3 (;-◁▷ distrib) `((`c_1 ◁ `b` ▷ c_2`); `c_3`) = ((`c_1`; `c_3`)` ◁ `b` ▷ `(`c_2`; `c_3`))`

3.1 Revising Laws Valid only for Copy Semantics

There are laws valid for copy semantics that, in general, are not valid for reference semantics. In what follows we discuss the problems that arise in some of these laws and we introduce a revised law valid for reference semantics. A more complete set of such laws can be found in [14]. In all of them the extension is performed by adding a side condition, in a similar way to the laws presented here.

The sequential composition of two assignments to the same list of variables is easily combined to a single assignment, when copy semantics is considered.

Law 4 (combine assignments) `(le := e; le := f) = (le := f[e/le])`

The notation `f[e/le]` denotes the substitution of `le` by the free occurrences of `e` in `f`. In the context of reference semantics, this law does not hold in general when `le` is a reference to an object. Assuming that `le` is a reference, we consider two kinds of assignments, depending on the form of `le`. If `le` is a variable `x`, the law is still valid, as it corresponds to two consecutive associations of objects to `le`, which is equivalent to the last association. However, if `le` is of form `le.a`, the following situation might happen. Assume `x` and `y` reference the same object, say an account with balance 2; the sequence of assignments

```
x.balance := y.balance + 1; x.balance := y.balance + x.balance
```

increases the balance attribute to 6, whereas the combined assignment updates it to 5 (`x.balance := y.balance + (y.balance + 1)`).

To be valid for reference semantics we must introduce conditions on the application of the law. In this case, if there is no aliasing between variables of `le` and variables of `f`, the law is sound. We introduce the function `nosh(le,f)` to capture this fact, where `nosh` stands for 'no sharing'; it yields true in the absence of aliasing. Thus, for reference semantics the law is as follows.

Law 5 (revised combine assignments)
`If nosh(le,f) then (le := e; le := f) = (le := f[e/le]).`

The computation of `nosh` depends on the context of the transformation and can be performed through a static analysis of the relevant program context, based, for example, on some ideas of [16, 17]. Broadly a graph G is constructed, where the nodes represent left-expressions (in the form x or $le.a$) and two nodes, say u and v, are connected by an edge if and only if an alias between u and v is introduced

by an assignment or a parameter passing mechanism. A depth-first search rooted in a given variable, say w, collects all variables potentially aliased with w. Thus, `nosh(le,f)` holds if `f` cannot be reached by applying a depth-first search rooted in `le` in G (see details in [14]). This analysis identifies all possibilities of aliasing and may be stronger than necessary to guarantee soundness. For example, aliasing of variables introduced by a branch of a conditional command that never executes are recorded in G, but does not really occur.

If two consecutive assignments do not share data, the order does not matter. For copy semantics the following law is valid.

Law 6 (order independent assignments)
`(le1 := e; le2 := f) = (le2 := f; le1 := e),`
`provided le1 ∉ free(f) and le2 ∉ free(e).`

This law presents a similar problem as the one related to Law 4 (combine assignments). If $\texttt{le}_1$ and $\texttt{le}_2$ have the form `x`, this corresponds to the attachment of a new value to these variables and the effect is the same as for variables of primitive types. However, if either $\texttt{le}_1$ or $\texttt{le}_2$ is of the form `le.a` the conditions $\texttt{nosh(le}_1\texttt{,f)}$ and $\texttt{nosh(le}_2\texttt{,e)}$ are required. The modified law for reference semantics is as follows.

Law 7 (revised order independent assignments)
`If (nosh(le1,f) ∧ nosh(le2,e) ∧ le1 ∉ free(f) ∧ le2 ∉ free(e)) then`
`(le1 := e; le2 := f) = (le2 := f; le1 := e).`

Assignment distributes rightwards through a conditional; the next law is also valid only for copy semantics.

Law 8 (assignment-◁▷ distrib) `If le ∉ free(b) then`
`((le := e); c1◁ b ▷c2) = ((le := e; c1)◁ b ▷(le := e; c2))`

If `le` is of the form `x`, this corresponds to an association of a new value to `le` and as `le` does not occur in `b`, the assignment can safely take place after `b` evaluation. However, if `le` is of the form `le.a` the condition `nosh(le,b)` is required. The modified law for reference semantics is presented below.

Law 9 (revised assignment-◁▷ distrib 1) `If (nosh(le,b) ∧ le ∉ free(b))`
`then ((le := e); c1◁ b ▷c2) = ((le := e; c1)◁ b ▷(le := e; c2)).`

However, a particular case of this law is valid for both copy and reference semantics. Consider an assignment `le := e`. Assuming that `le` is an object identifier, this creates an aliasing between `le` and `e`. Such an assignment distributes leftwards through a conditional, if the condition contains occurrences of `e`, but no further aliasing with `le`.

Law 10 (revised assignment-◁▷ distrib 2) `If nosh(le,b[null]) then`
`((le := e); c1◁ b[e] ▷c2) = ((le := e; c1)◁ b[e] ▷(le :=e; c2)).`

The notation `b[e]` means that `b` might contain occurrences of `e`. The condition `nosh(le,b[null])` captures that there is no sharing between `le` and `b`; null is used in place of `e` (`b[null]`) to check that `le` has no further aliasing with `b`.

3.2 Laws Valid only for Reference Semantics

The introduction of aliasing motivates the investigation of new laws. In this section we introduce laws valid only in a context of reference semantics. We are also considering as an implicit condition that `le` refers to an object, so it is not a variable of a primitive type. We use the notation $\texttt{le}_\texttt{o}$ to remark this fact.

Permutation of assignments are possible in some situations.

Law 11 (assignment permutation)
$(\texttt{c; le}_\texttt{o} := \texttt{e}) = (\texttt{le}_\texttt{o} := \texttt{e; c})$, provided $\texttt{le}_\texttt{o} \notin \texttt{free(c)}$.

This law is not valid in general for copy semantics as the value of `e` might be altered in `c` and, in this case, $\texttt{le}_\texttt{o}$ will hold the old value on the right-hand side of the law. This problem does not arise in reference semantics because $\texttt{le}_\texttt{o}$ and `e` are aliased and any change in `e` on the right-hand side will also reflect in $\texttt{le}_\texttt{o}$. Even in the case that $\texttt{le}_\texttt{o}$ is previously aliased with an object that occurs in `c`, say `z`, the law is still valid. On both sides of the law the value of $\texttt{le}_\texttt{o}$ is `e` and the aliasing with `z` is broken. Moreover, as $\texttt{le}_\texttt{o}$ is not free in `c`, `z` cannot be indirectly updated by $\texttt{le}_\texttt{o}$ on the left-hand side of the law. Thus, the final value of `z` is not affected by the assignment permutation.

If two variables are aliased, it does not matter which one is chosen to use.

Law 12 (assignment seq substitution) `If preserves(`$\texttt{le}_\texttt{o}$`, e, c) then` $(\texttt{le}_\texttt{o}\texttt{:= e; c}) = (\texttt{le}_\texttt{o} \texttt{:= e; c[e/le}_\texttt{o}\texttt{]})$.

The function `preserves(`$\texttt{le}_\texttt{o}$`, e, c)` holds if the alias between $\texttt{le}_\texttt{o}$ and `e` is preserved in `c`. To guarantee the preservation of the aliasing of $\texttt{le}_\texttt{o}$ and `e` the code is analyzed and it is checked if $\texttt{le}_\texttt{o}$ does not appear neither in the left-hand side of assignments nor as a result of value-result parameter of a method call in `c`.

To see why this law is not valid for copy semantics, consider the case in which `e` is `z` and `c` is a method call `z.m()`, where `m` is a method that updates an attribute of the object referenced by `z`. By applying the law we have

```
(y := z; z.m()) = (y := z; y.m()).
```

If we consider reference semantics, this equality is valid as `z` and `y` refer to the same object. However, in the context of copy semantics this is not the case; on the left-hand side the object referenced by `z` is updated, whereas on the right-hand side the object referenced by `y` is modified and they are distinct objects.

As mentioned before, alias can occur through assignments and parameter passing. Thus, there are analogous laws to the previous one that substitute the assignment by a method call to `m` through which the aliasing between $\texttt{le}_\texttt{o}$ and `e` occurs. There are several situations to consider. For example, the method call could be $\texttt{le}_\texttt{o}\texttt{.m}(\emptyset,\emptyset\texttt{,e})$ and the body of `m` includes an assignment `self := e`. In another context, both $\texttt{le}_\texttt{o}$ and `e` could be passed as result or value-result arguments of `m` and the body of `m` includes an assignment `x := y`, where `x` and `y` are the formal parameters bound to $\texttt{le}_\texttt{o}$ and `e`, respectively. The aliasing between $\texttt{le}_\texttt{o}$ and `e` may occur in a sequence of nested method calls, and so on.

To abstract all situations in which aliasing could happen, we introduce the notation $\texttt{c[alias(le}_\texttt{o}\texttt{,e)]}$, meaning that `c` is a command that establishes an alias between $\texttt{le}_\texttt{o}$ and `e`. The previous law can be generalized as follows.

Law 13 (aliasing seq substitution) If preserves(le_o, e , c_2) then
(c_1[alias(le_o,e)]; c_2) = (c_1[alias(le_o,e)]; c_2[e/le_o]).

If two variables le_{o1} and le_{o2} are aliased, sequential assignments of le_{o1} to le_{o2} and vice-versa have no effect.

Law 14 (redundant assignment)
(c[alias(le_{o1},le_{o2})]) = (c[alias(le_{o1},le_{o2})]; le_{o1} := le_{o2})
= (c[alias(le_{o1},le_{o2})]; le_{o2} := le_{o1})

As le_{o1} and le_{o2} reference the same object, the last assignment after executing c can be regarded as the skip command in the previous law and, therefore, can be removed. However, for copy semantics, the final assignment cannot be eliminated, otherwise the updating of le_{o1} (or le_{o2}) is not perceived by le_{o2} (or le_{o1}).

4 Application: Code Refactoring

The proposed laws can be used to formalize refactorings and, more generally, to improve code structure. For example, Rule 1 (extract/inline method) expresses the Extract/Inline refactoring [13], which can be proved using some of the laws presented in Sect. 3 (see [14]). The extract method is used to group code fragments as a new method. The inline method does the opposite task.

Let cds be the set of declared classes of program P. Let c be the main command of P. Let A and C be classes of cds; ads, pds and mts the attribute, parameter and method declarations, respectively.

Rule 1 (extract/inline method)
class A extends C{ads;m_1(pds_1){c_1[c_2[a]]};mts}
=
class A extends C{ads;m_1(pds_1){c_1'};m_2(pds_2){c_2[α(pds_2)];mts'}

where

m_2 does not appear in c nor in mts, for any B $\preceq$ A,
c_1[c_2[a]] denotes that c_2[a] is a fragment of code of c_1,
c_1' $\stackrel{def}{=}$ c_1[self.m_2(a)/c_2(a)], mts' $\stackrel{def}{=}$ mts[c_2[a]/self.m_2(a)],
a is the set of variables of c_2, not including attributes of class A;

provided

(1) parameters in pds_2 must have the same types as variables in a;
(2) method m_2 is not declared in mts nor in any superclass or subclass
of A in cds.

In this rule, from left to right, the fragment of code represented by c_2 is extracted from all methods in A (m_1 and mts) and a new method with body c_2 is created. The modified methods then call the created method. The reverse action is done when applying the rule from right to left.

To apply this transformation we consider some issues, captured by the side conditions associated to the rule. The first condition is obviously necessary to guarantee a type safe matching, and is required in both directions of the rule application. The second condition is necessary for the application from left to right, requiring that the name $\mathtt{m}_2$ be fresh. The third condition is necessary for the application from right to left, to guarantee that only methods of class `A` calls $\mathtt{m}_2$ and thus it is possible to inline its body.

Cornélio [12] gives a formalization of this rule for copy semantics. In his formalization, types of variables in `a` must be of a basic type, otherwise changes of objects in $\mathtt{m}_2$ may not be reflected in variables of $\mathtt{m}_1$. This is a severe restriction in the application of the general refactoring proposed by Fowler. In the context of reference semantics, we are able to deal with the general case. Similar refactorings can be formalized in an analogous way.

To illustrate an application of code restructure that uses the above refactoring and some of the proposed laws, consider program fragment that searches and updates an element of an array; this is intentionally unstructured, combining the actions related to search and update into a single method as shown in Fig. 1. The goal is to transform this program into the one in Fig. 2, in which a method search has been introduced to separate the tasks of updating and searching in different methods.

```
class BasicEntity {
    private T1 at1; T2 at2;
    method update(T2 m ; ∅; ∅){at2 := m}; getat1(∅; T1 n; ∅){n:=at1}}
class Application {
    private BasicEntity data[]; -- assume the last element is null
    method updateApplication(T1 num, T2 value; T2 reply ; ∅)
    {   Int i = 1;
        Int len, number; Bool stop;
        len := data.length(); stop := false;
        SKIP ◁ data[i] == null ▷ {
            ((¬ stop) and (i<=len)) ∗
                {data[i].getat1(∅, number, ∅);
                stop := true ◁ number = num ▷ i := i + 1}}
        {reply := "Failed Updating"} ◁ data[i] == null ▷
           {data[i].update(value, ∅, ∅); reply:= "Successful Updating"}}}
```

Fig. 1. Searching and updating an element of an array in a single method

To improve the code structure we begin with the example of Fig. 1 and by applying some of the laws introduced in Sect. 3, we reach the code depicted in Fig. 2. As each law is proved considering the rCOS semantics (see [14] for details), the code transformation is sound. Thus, initially we apply Law 1 (var elim) to introduce a declaration of `obj` in method `updateApplication`. After that, we apply Law 2 (var final value) to introduce the alias between `obj` and `data[i]` (see Fig. 3).

```
class BasicEntity {... }
class Application {
    private BasicEntity data[]; -- assume the last element is null
    method search(T1 num; BasicEntity obj; ∅) {
        Int i = 1;
        Int len, number; Bool Stop
        len := data.length(); stop := false;
        SKIP ◁ data[i] == null ▷ {
            ((¬ stop) and (i<=len)) ∗
                {data[i].getat1(∅, number, ∅);
                stop := true ◁ number = num ▷ i := i + 1}}
        obj:=data[i]}
    method updateApplication(T1 num, T2 value; T2 reply ; ∅) {
        BasicEntity elem;
        self.search(num, elem, ∅);
        {reply := "Failed Updating"} ◁ elem == null ▷
          {elem.update(value, ∅, ∅); reply:= "Successful Updating" }}}
```

Fig. 2. An example of searching and updating an element of an array

```
class BasicEntity {...} class Application {...
    method updateApplication(T1 num, T2 value; T2 reply ; ∅)
    {  ...  BasicEntity obj; ...
           {reply := "Failed Updating"} ◁ data[i] == null ▷
                {data[i].update(value); reply:= "Successful Updating"}}}
            obj:= data[i];
```

Fig. 3. Applying laws 1 (var elim) and 2 (var final value) in the case study of Fig. 1

```
class BasicEntity {...} class Application {...
    method updateApplication(T1 num, T2 value; T2 reply ; ∅)
    {  ... BasicEntity obj; ...
           { obj := data[i];
            reply := "Failed Updating"} ◁ data[i] == null ▷
                {data[i].update(value); reply:= "Successful Updating"}}}
```

Fig. 4. Applying laws 3 (;-◁▷ distrib), 7 (revised order independent assignment), 11 (assignment permutation) and 10 (revised assignment-◁▷ 2) in the example of Fig. 3

Then we apply Law 3 (;-◁▷ distrib) to move the assignment to `obj` to inside the conditional. Now, we apply Law 7 (revised order independent assignment) twice to permute the assignment to `obj`. In this case, as `data[i]` and `reply` are of different types, `nosh(reply,data[i])` is trivially true, as well as `nosh(obj,k)`, where `k` is the string data. Moreover, `obj` and `reply` are not on the right-hand side expressions of any of these assignments. Next, we use Law 11 (assignment permutation) to permute the assignment to `obj`. Then, we use Law 10 (revised assignment-◁▷ 2) to permute again the assignment to `obj`. Fig. 4 shows the result-

ing code. Next, we apply Law 12 (assignment seq substitution) to replace `data[i]` with `obj`. Finally, we apply Rule 1 (extract/inline method) to extract the `search` method achieving the program depicted in Fig. 2.

5 Conclusions

Programming laws for imperative and concurrent languages are well established and have been proven useful in the design of applications of program transformation like compilers [18] and hardware/software partitioning [19]. The major contribution of this work is to provide a set of laws for object-oriented languages based on a reference semantics. We have considered laws already proposed for copy semantics, that are not valid for reference semantics, and for each of these laws, we have proposed a revised version for reference semantics. Furthermore, this work introduces new laws valid only for reference semantics. We are not aware of any other result in this direction.

A common criticism of the algebraic style is that merely postulating algebraic laws can give rise to complex and unexpected interactions between programming construction. This can be avoided by the verification of the laws in a mathematical model. Our laws have been proved sound with respect to rCOS semantics, as illustrated in the appendix.

The full set of laws valid both for copy and reference semantics, along with the revised laws of copy semantics, presented in [14], is complete in the sense that it is sufficient to reduce an arbitrary program to a normal form expressed in a restricted subset of the language. The reduction strategy proposed in [11] can be adapted straightforwardly. A future work is to extend this strategy by considering also the laws valid only for reference semantics here introduced.

Although the case study presented here is simple, it illustrates the benefits of using algebraic laws to improve code structure, in a sound way, in the presence of aliasing. To develop more elaborate case studies is an important future work. These case studies will be particularly useful to validate and possibly extend the proposed set of laws.

In a context of reference semantics, data refinement is challenging. Although data refinement can be proved directly in the semantics, this tends to be a laborious task. We intend to investigate laws for data refinement in the style of Morgan's work [20], based on approaches to *confinement* [21]. Moreover, if we consider confinement, the possibilities of aliasing among variables are restricted. We also plan to investigate an alternative presentation of the laws considering confined programs, rather than using the explicit side conditions based on `nosh`.

References

1. Abadi, M., Leino, K.R.M.: A logic of object-oriented programs. In Bidoit, M., Dauchet, M., eds.: TAPSOFT'97: Theory and Practice of Software Development. Volume 1214 of LNCS, Springer-Verlag (1997)
2. Müller, P., Poetzsch-Heffter, A.: Formal specification techniques for object-oriented programs. In Jarke, M., Pasedach, K., Pohl, K., eds.: Informatik 97: Informatik als Innovationsmotor, Springer-Verlag (1997)

3. Cavalcanti, A.L.C., Naumann, D.A.: A weakest precondition semantics for refinement of object-oriented programs. IEEE Trans. on Soft. Eng. **26** (2000) 713–728
4. He, J., Li, X., Liu, Z.: rCOS: A refinement calculus of object systems. Theoretical Computer Science **365** (2006) 109–142
5. He, J., Hoare, C.A.R.: Unifying Theories of Programming. Prentice-Hall (1998)
6. Hoare, C.A.R., Hayes, I.J., He, J., Morgan, C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M., Sufrin, B.A.: Laws of programming. Commun. ACM **30** (1987) 672–686
7. Bird, R., de Moor, O.: Algebra of Programming. Prentice-Hall (1997)
8. Seres, S., Spivey, J.M., Hoare, C.A.R.: Algebra of logic programming. ICPL (1999)
9. Roscoe, A.W., Hoare, C.A.: The laws of **occam** programming. Theoretical Computer Science **60** (1988) 177–229
10. Leino, K.R.M.: Recursive object types in a logic of object-oriented programming. Nordic Journal of Computing **5** (1998) 330–360
11. Borba, P., Sampaio, A., Cavalcanti, A.L.C., Cornélio, M.: Algebraic reasoning for object-oriented programming. Sci. Comput. Programming **52** (2004) 53–100
12. Cornélio, M.: Refactoring as Formal Refinements. PhD thesis, Federal University of Pernambuco, Centro de Informática, UFPE, Recife, Brazil (2004)
13. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (2000)
14. Silva, L., Sampaio, A., Liu, Z.: Laws of object-oriented languages: copy versus reference semantics. TR www.cin.ufpe.br/~lmas/report-laws.pdf, UFPE (2007)
15. Noble, J., Vitek, J., Potter, J.: Flexible alias protection. In: European Conference on Object-Oriented Programming (ECOOP). LNCS, Springer-Verlag (1998)
16. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA (2002)
17. Jackson, D., Rollins, E.: Abstractions of program dependencies for reverse engineering. In: Proc. ACM SIGSOFT Conf. on Foundations of Soft. Eng. (1994)
18. Sampaio, A.: An Algebraic Approach to Compiler Design. Volume 4 of AMAST Series in Computing. World Scientific (1997)
19. Silva, L., Sampaio, A., Barros, E.: A constructive approach to hardware/software partitioning. Formal Methods In System Design **24** (2004) 45–90
20. Morgan, C.: Programming from Specifications. 2nd edn. Prentice-Hall (1994)
21. Banerjee, A., Naumann, D.A.: Ownership confinement ensures representation independence for object-oriented programs. Journal of the ACM **52** (2005) 894–960

A Proof of Law 11 (*assignment permutation*)

The proofs of this law use the following auxiliary results of designs and predicates, extracted from [5].

Law 15 (*predicate 1)* `true;false = false = false;true`

Law 16 (*;-predicate)*

`P(v');Q(v)` $\stackrel{def}{=}$ $\exists$ v_0 $\bullet$ `P(`v_0`)`$\wedge$ `Q(`v_0`), provided out`α`P = in`α`'Q =`{`v'`}

outαP is the output variables `v'` of `P` and inαQ is the input variables of `Q`.

The normal combinators of the programming language have exactly the same meaning as operators on the single predicates as they have on the double predicates of the refinement calculus.

Theorem 1. *Let P_1, P_2, Q_1 and Q_2 be predicates*

$$(P_1 \vdash Q_1) \lhd b \rhd (P_2 \vdash Q_2) = (P_1 \lhd b \rhd P_2) \vdash (Q_1 \lhd b \rhd Q_2)$$
$$(P_1 \vdash Q_1); (P_2 \vdash Q_2) = (\neg(\neg P_1; true) \wedge \neg(Q_1; \neg P_2)) \vdash (Q_1; Q_2)$$

An immediate consequence of this theorem is given below.

Corollary 1. $(true \vdash Q_1); (true \vdash Q_2) = true \vdash (Q_1; Q_2)$

Proof.

$$\begin{array}{ll} & LHS \\ = & \langle Theorem\ 1 \rangle \\ & (\neg(\neg true; true) \wedge \neg(Q_1; \neg true)) \vdash (Q_1; Q_2) \\ = & \langle predicate\ calculus\ and\ Law\ 15 (predicate\ 1) \rangle \\ & (true \wedge true) \vdash (Q_1; Q_2) \\ = & \langle boolean\ algebra \rangle \\ & RHS \end{array}$$

To perform the proof of Law 11 (*assignment permutation*) we have to consider all possible forms of command $\mathtt{c_1}$. Here we prove the case when $\mathtt{c_1}$ is a method call, as this situation happens in the case study.

Proof.

$$\begin{array}{ll} & [\![e.m(v,\ \emptyset,\ \emptyset);\ le := e]\!] \\ = & \langle from\ UTP\ semantics \rangle \\ & [\![e.m(v,\ \emptyset,\ \emptyset)]\!]; [\![le := e]\!] \\ = & \langle semantics\ of\ method\ call\ and\ assignment\ \rangle \\ & (true \vdash \exists C \in cname \bullet \mathrm{type}(e) = C \wedge [\![var\ T\ x = v]\!]; [\![var\ C\ self = e]\!]; \\ & [\![Execute(C.m)]\!]; [\![end\ self,x]\!]); (true \vdash \forall le \bullet \exists \overline{le} \bullet \overline{le}' = \langle \mathrm{value}(e) \rangle.\mathrm{tail}(\overline{le})) \\ = & \langle semantics\ of\ variable\ declaration\ and\ undeclaration;\ semantics\ of\ method\ call; \\ & Corollary\ 1; Law\ 16\ (;-predicate) and\ predicate\ calculus \rangle \\ & (true \vdash \exists C \in cname \bullet \mathrm{type}(e) = C \wedge \\ & \exists v : T \bullet (\forall x, self \bullet \exists \overline{x}, \overline{self} \bullet \exists \overline{x}_0, \overline{self}_0 \bullet \overline{x}_0 = \langle \mathrm{value}(v) \rangle.\mathrm{tail}(\overline{x}) \wedge \\ & \overline{self}_0 = \langle \mathrm{value}(e) \rangle.\mathrm{tail}(\overline{self}) \wedge \pi' = \{\langle r, C, \sigma_0 \bigoplus \{a \mapsto v\} \rangle | r = \mathrm{ref}(\mathrm{value}(e))\} \wedge \\ & \overline{self}' = \mathrm{tail}(\overline{self}_0) \wedge \overline{x}' = \mathrm{tail}(\overline{x}_0))); (true \vdash \forall le \bullet \exists \overline{le} \bullet \overline{le}' = \langle \mathrm{value}(e) \rangle.\mathrm{tail}(\overline{le})) \\ = & \langle predicate\ calculus,\ Corollary\ 1;\ Law\ 16\ (;-predicate) \rangle \\ & true \vdash (\exists C \in cname \bullet \mathrm{type}(e) = C \wedge \\ & \exists v : T \bullet (\forall x, self \bullet \exists \overline{x}, \overline{self} \bullet \exists \overline{x}_0, \overline{self}_0 \bullet \overline{x}_0 = \langle \mathrm{value}(v) \rangle.\mathrm{tail}(\overline{x}) \wedge \\ & \overline{self}_0 = \langle \mathrm{value}(e) \rangle.\mathrm{tail}(\overline{self}) \wedge \pi' = \{\langle r, C, \sigma_0 \bigoplus \{a \mapsto v\} \rangle | r = \mathrm{ref}(\mathrm{value}(e))\} \wedge \\ & \overline{self}' = \mathrm{tail}(\overline{self}_0) \wedge \overline{x}' = \mathrm{tail}(\overline{x}_0)) \wedge \forall le \bullet \exists \overline{le} \bullet \overline{le}' = \langle \mathrm{value}(e) \rangle.\mathrm{tail}(\overline{le})) \\ = & \langle le\ is\ free\ in\ the\ previous\ predicate;\ predicate\ calculus \rangle \\ & true \vdash (\forall le \bullet \exists \overline{le} \bullet \overline{le}' = \langle \mathrm{value}(e) \rangle.\mathrm{tail}(\overline{le})) \wedge \\ & (\exists C \in cname \bullet \mathrm{type}(e) = C \wedge \exists v : T \bullet (\forall x, self \bullet \exists \overline{x}, \overline{self} \bullet \\ & \pi' = \{\langle r, C, \sigma_0 \bigoplus \{a \mapsto v\} \rangle | r = \mathrm{ref}(\mathrm{value}(e))\} \wedge \overline{self}' = \overline{self} \wedge \overline{x}' = \overline{x})) \\ = & \langle Law\ 16 (;-predicate);\ Corollary\ 1;\ \ Theorem\ 1\ and\ predicate\ calculus \rangle \\ & [\![le := e]\!]; [\![e.m(v,\ \emptyset,\ \emptyset)]\!] \\ = & \langle from\ UTP\ semantics \rangle \\ & [\![le := e; e.m(v,\ \emptyset,\ \emptyset)]\!] \quad \square \end{array}$$